

WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

A MODERN APPROACH TO
COMPUTER SYSTEMS FOR LINEAR PROGRAMMING

988-78

March 1978

Robert Fourer
Michael J. Harrison

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

Working Paper

Alfred P. Sloan School of Management

Center for Computational Research
in Economics and Management Science

A MODERN APPROACH TO
COMPUTER SYSTEMS FOR LINEAR PROGRAMMING

988-78

March 1978

Robert Fourer
Michael J. Harrison

Massachusetts Institute of Technology
50 Memorial Drive
Cambridge, Massachusetts 02139

Research for this paper was supported by grants to the Center from the
National Science Foundation.

Address inquiries to the authors as follows:

Robert Fourer
Department of Operations Research
Stanford University
Stanford, CA 94305

Michael J. Harrison
National Opinion Research Center
University of Chicago
6030 South Ellis Avenue
Chicago, IL 60637

ABSTRACT

Popular computer packages for linear programming do not differ much in concept from ones devised ten or twenty years ago. We propose a modern LP system -- one that takes advantage of such (relatively) new ideas as high-level languages, interactive and virtual operating systems, modular design, and hierarchical file systems. Particular topics include: computer languages that describe optimization models algebraically; specialized editors for models and data; modular algorithmic codes; and interactive result reporters. We present specific designs that incorporate these features, and discuss their likely advantages (over current systems) to both research and practical model-building.

CONTENTS

A MODERN APPROACH TO COMPUTER SYSTEMS FOR LINEAR PROGRAMMING . .	1
1 THE TRADITIONAL APPROACH	5
The algorithm	5
Matrix generators	5
Report writers	9
Present-day systems	10
2 A MODERN APPROACH	11
Describing the model in algebraic form	11
Using the computer	15
Outline of a modern LP	16
3 A MODELING LANGUAGE	19
Sets	19
Parameters	23
Variables	25
Constraints	25
Objectives	27
XML syntactic forms	27
4 PREPARING AN LP FOR SOLUTION	29
Model organization: a hierarchical arrangement	29
Model creation: the model editor	32
Model creation, continued: the data editor	35
Model translation: the model translator	37
5 SOLVING LINEAR PROGRAMS	39
External design of an LP solver	39
Internal design of an LP algorithm	44
6 REPORTING RESULTS	49
Storing solutions	49
Tabulating results	51
Tabulating in AMPS: XSR	54
Graphical presentation of results	55
REFERENCES	57

FIGURES

1a	A sample LP in algebraic form	2
1b	Sample data for the algebraic LP	3
1c	The same sample LP in matrix form	3
2	An MPS deck for the matrix of Figure 1c	6
3	A matrix-generator program for the model of Figure 1	8
4	Linear program of Figure 1a expressed in the modeling language XML	20
5	Elements of an XML component	28
6	Hierarchical arrangement of LP information	30
7	XME model-editing commands	34
8	Conceptual file hierarchy for one case of one model	40

A MODERN APPROACH TO COMPUTER SYSTEMS FOR LINEAR PROGRAMMING

The success of linear programming lies in two facts. First, many and diverse practical problems require (or can be formulated as) minimization of a linear combination of variables, constrained by linear equalities and inequalities. Second, almost every such problem can be solved routinely and efficiently by use of a single general algorithm, the simplex method.

Corresponding to these two observations are two necessary forms of a linear program. When a modeler builds an LP, he expresses it in its natural "algebraic form": he defines constants and variables of the problem; writes an objective as an arithmetic expression that is linear in the variables; and writes the constraints as equalities or inequalities between linear expressions. The simplex algorithm, by contrast, needs the problem in its "matrix form": a series of column vectors, each column being the coefficients of one variable. (Figure 1 contrasts algebraic and matrix form, for a simple LP.)

Modelers can't work efficiently with matrix form, and the algorithm can't employ algebraic form. As a result, computer systems for linear programming face three tasks:

- (1) Translate the model from modeler's (algebraic) to algorithm's (matrix) form.
- (2) Solve, using the simplex method.
- (3) Report the solution in modeler's (algebraic-form) terminology.

We summarize below two approaches to these tasks: the traditional one (Section 1), and what we see as a "modern" one (Section 2). Subsequent sections explore the modern approach in further detail.

Figure 1a. A sample LP in algebraic form. You have a set of production activities requiring various amounts of raw materials. You have to buy stocks of raw materials now for the next T production periods; but your warehouses have limited capacity, and there is a storage cost each period. Total production each period is also limited. What set of production activities yields the greatest expected net profit?

Given: P set of products
 R set of raw materials

and: T number of periods
 M maximum total unit production per period
 a_{ij} units of raw material i required to produce 1 unit of product j ; $i \in R, j \in P$
 b_i maximum initial stock of raw material i ; $i \in R$
 c_{jt} expected gross profit per unit of product j in period t ; $j \in P, t=1, \dots, T$
 d_i storage cost per unit of raw material i ; $i \in R$

Define variables: x_{jt} units of product j manufactured in period t ; $j \in P, t=1, \dots, T$
 s_{it} stock of raw material i at beginning of period t ; $i \in R, t=1, \dots, T+1$

Maximize: $\sum_{t=1}^T (\sum_{j \in P} c_{jt} x_{jt} - \sum_{i \in R} d_i s_{it})$

Subject to: $s_{i1} \leq b_i, \quad i \in R$
 $s_{i,t+1} = s_{it} - \sum_{j \in P} a_{ij} x_{jt}, \quad i \in R, t=1, \dots, T$
 $\sum_{j \in P} x_{jt} \leq M, \quad t=1, \dots, T$
 $x_{jt} \geq 0, \quad j \in P, t=1, \dots, T$
 $s_{it} \geq 0, \quad i \in R, t=1, \dots, T+1$

Figure 1b. Sample data for the algebraic LP. There are two classes of raw material (SCRAP, NEW) used to make products of three qualities (LOW, MED, HIGH).

$$P = \{LOW, MED, HIGH\}$$

$$R = \{SCRAP, NEW\}$$

$$T = 3$$

$$M = 40$$

$$a = \begin{array}{c} \begin{array}{ccc} & \begin{array}{ccc} & 1 & 2 & 3 \end{array} \\ \begin{array}{ccc} LOW & MED & HIGH \end{array} \\ \begin{bmatrix} 5 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix} \end{array} \begin{array}{l} SCRAP \\ NEW \end{array} \end{array} \quad c = \begin{array}{c} \begin{bmatrix} 25 & 20 & 10 \\ 50 & 50 & 50 \\ 55 & 60 & 80 \end{bmatrix} \begin{array}{l} LOW \\ MED \\ HIGH \end{array} \end{array}$$

$$b = \begin{array}{c} \begin{bmatrix} 400 \\ 275 \end{bmatrix} \begin{array}{l} SCRAP \\ NEW \end{array} \end{array} \quad d = \begin{array}{c} \begin{bmatrix} .5 \\ 2 \end{bmatrix} \begin{array}{l} SCRAP \\ NEW \end{array} \end{array}$$

Figure 1c. The same sample LP in matrix form. The last column is the right-hand-side constant, after all variables have been transposed to the left. The constraints $x_{jt} \geq 0$ and $s_{it} \geq 0$ are enforced implicitly by the simplex method, and so are not represented in the matrix.

$$\begin{array}{c} \begin{array}{cccccccccccccccccccc} x_{LOW,1} & x_{LOW,2} & x_{LOW,3} & x_{MED,1} & x_{MED,2} & x_{MED,3} & x_{HIGH,1} & x_{HIGH,2} & x_{HIGH,3} & s_{SCRAP,1} & s_{SCRAP,2} & s_{SCRAP,3} & s_{SCRAP,4} & s_{NEW,1} & s_{NEW,2} & s_{NEW,3} & s_{NEW,4} & RHS \end{array} \\ \left[\begin{array}{cccccccccccccccccccc} 25 & 20 & 10 & 50 & 50 & 50 & 55 & 60 & 80 & -.5 & -.5 & -.5 & & -2 & -2 & -2 & & 0 \\ & & & & & & & & & 1 & & & & & & & & 400 \\ 5 & & & 3 & & & 1 & & & -1 & 1 & & & & & & & 0 \\ & 5 & & & 3 & & & 1 & & & -1 & 1 & & & & & & 0 \\ & & 5 & & & 3 & & & 1 & & & -1 & 1 & & & & & 0 \\ & & & & & & & & & & & & & 1 & & & & 275 \\ 1 & & & 2 & & & 3 & & & & & & & -1 & 1 & & & 0 \\ & 1 & & & 2 & & & 3 & & & & & & & -1 & 1 & & 0 \\ & & 1 & & & 2 & & & 3 & & & & & & & -1 & 1 & 0 \\ 1 & & & 1 & & & 1 & & & & & & & & & & & 40 \\ & 1 & & & 1 & & & 1 & & & & & & & & & & 40 \\ & & 1 & & & 1 & & & 1 & & & & & & & & & 40 \end{array} \right] \begin{array}{l} (max) \\ (\leq) \\ (=) \\ (=) \\ (=) \\ (\leq) \\ (=) \\ (=) \\ (=) \\ (\leq) \\ (\leq) \\ (\leq) \end{array} \end{array}$$

1 THE TRADITIONAL APPROACH

The Algorithm

The first concern of LP system designers was implementation and perfection of the simplex method. Hence early systems were built around the algorithmic code, or "optimizer". Input was a listing of the columns of the problem matrix; output was a vector of activities.

At this early stage the principles of operating systems, file systems, and interactive computing were primitive or unknown. Hence LP "systems" were essentially big programs that ran as batch jobs. They created their own operating environments and managed their own files.

Matrix generators

For problems of any size and complexity, specifying every matrix element by row and column number was hopelessly inefficient and error-prone. A more practical scheme for matrix input quickly arose: rows and columns were given unique names, typically of up to 8 letters and numbers. Each nonzero matrix element was then specified by giving a row name, a column name, and a value. A common input format for this arrangement was a "matrix deck" in some standard form such as MPS (Figure 2), OPTIMA, APEX-II, SHARE, or SESAME [5,15,23].

Still, translating an algebraic-form LP to a matrix deck required much repetitive and tedious work. It was therefore natural to try to use the computer to build the deck, from the user's instructions and data. The first programs for this purpose were written in FORTRAN and other general-purpose languages, and were somewhat successful with medium-sized or well-established applications [2].

For large complex models, however, and for ones subject to development and change, FORTRAN was still too cumbersome. The logical next step

Figure 2. An MPS deck for the matrix of Figure 1c. Appropriate row and column names have been invented. The constraints $s_{i1} \leq b_i$ are represented in the "BOUNDS" section at the end: this tells the algorithm to handle them as implicit bounds rather than as explicit constraints. Nonnegativity of all variables is assumed.

```

NAME          SAMPLE
ROWS
N  OBJ
E  BALSCR1
E  BALSCR2
E  BALSCR3
E  BALNEW1
E  BALNEW2
E  BALNEW3
L  CAP1
L  CAP2
L  CAP3
COLUMNS
XLOW1  BALSCR1  5          BALNEW1  1
XLOW1  CAP1     1          OBJ        25
XLOW2  BALSCR2  5          BALNEW2  1
XLOW2  CAP2     1          OBJ        20
XLOW3  BALSCR3  5          BALNEW3  1
XLOW3  CAP3     1          OBJ        10
XMED1  BALSCR1  3          BALNEW1  2
XMED1  CAP1     1          OBJ        50
XMED2  BALSCR2  3          BALNEW2  2
XMED2  CAP2     1          OBJ        50
XMED3  BALSCR3  3          BALNEW3  2
XMED3  CAP3     1          OBJ        50
XH1H1  BALSCR1  1          BALNEW1  3
XH1H1  CAP1     1          OBJ        75
XH1H2  BALSCR2  1          BALNEW2  3
XH1H2  CAP2     1          OBJ        80
XH1H3  BALSCR3  1          BALNEW3  3
XH1H3  CAP3     1          OBJ        100
SSCR1  BALSCR1  -1         OBJ        -1
SSCR2  BALSCR1  1          BALSCR2  -1
SSCR2  OBJ      -1.5
SSCR3  BALSCR2  1          BALSCR3  -1
SSCR3  OBJ      -1.5
SSCR4  BALSCR3  1
SNEW1  BALNEW1  -1         OBJ        -2
SNEW2  BALNEW1  1          BALNEW2  -1
SNEW2  OBJ      -2
SNEW3  BALNEW2  1          BALNEW3  -1
SNEW3  OBJ      -2
SNEW4  BALNEW3  1
RHS
RHSIDE  CAP1      40          CAP2      40
RHSIDE  CAP3      40
BOUNDS
UP  INTRND  SSCR1      400
UP  INTRND  SNEW1      275
ENDATA

```

was a "matrix generator" (MG) system designed specifically for creating matrix decks. An MG was operated by writing a program in a specially-designed language. The program first declared sets of indices and tables of numerical data; it indicated how names of rows and columns were to be formed; and, for each column, it specified the non-zero matrix elements. Figure 3 is an example of a program for the MaGen matrix generator [11]. Other MGs designed along the same lines include OMNI [12], GAMMA [3], and APEX-II MRG [5] (all close cousins of MaGen), DATAFORM [18] and DATAMAT [21].

Matrix generators offered numerous advantages to hand-coding of matrix decks:

- ¶ They provided for organizing and storing the problem data.
- ¶ They made it easy to enforce a uniform scheme for naming rows and columns.
- ¶ They let a user change model structure or model data with much less work -- often it sufficed just to change a few statements in the MG program and run it again.
- ¶ They reduced clerical errors; moreover, the logical structure of the MG program made mistakes in formulation somewhat easier to catch.

It is safe to say that many large and ambitious models were practical only through use of matrix generators.

Nevertheless, the common design of matrix generators had serious limitations:

- ¶ Like traditional optimizers, MGs ran in batch mode and required their own file structure and environment.
- ¶ The algebraic form of a model still had to be translated by hand, to an MG program. Writing this program was seldom easy, and was impossible till one mastered the special MG programming language.

Figure 3. A matrix-generator program for the model of Figure 1, in the MaGen language.

```

GENERATE,I
DICTIONARY
  CLASS I      TIMES (PERIODS)
  1,2,3
  CLASS U      TIMES PLUS LAST ONE
  1,2,3,4
  CLASS U      TIMES MINUS ONE
  0,1,2,3
  CLASS PRD    LOW QUALITY
  LOW          MEDIUM QUALITY
  MED          HIGH QUALITY
  HIH
  CLASS RAW    SCRAP MATERIAL
  SCR          NEW MATERIAL
  NEW

DATA
  TABLE A      INPUT-OUTPUT COEFFICIENTS
    LOW MED HIH
    SCR 5 3 1
    NEW 1 2 3

  TABLE R      MAXIMUM INITIAL STOCKS
    MAX
    SCR 400
    NEW 275

  TABLE C      EXPECTED GROSS PROFITS
    1 2 3
    LOW 25 20 10
    MED 50 50 50
    HIH 55 60 80

  TABLE D      STORAGE COSTS
    STOR
    SCR .5
    NEW 2.

*
  TABLE M      MAX PRODUCTION PER PERIOD
    MAX
    40

  *
  NAME          COPY          SAMPLE
  FORM ROW ID
  OBJ = OBJ
  BAL(RAW)(T) = FIX
  CAP(T) = MAX

  *
  COLUMNS      COPY
  FORM VECTOR X(PRD)(T)
  BAL(RAW)(T) = TABLE A((PRD),(RAW))
  OBJ = TABLE C(T),(PRD))
  CAP(T) = 1
  FORM VECTOR S(RAW)(V)
  BAL(RAW)(V) = -1 EXCEPT V = 4
  BAL(RAW)(U/V) = 1 EXCEPT V = 1
  OBJ = -TABLE D(STOR,(RAW)) EXCEPT V = 4

  *
  RHS           COPY
  FORM VECTOR RHSIDE
  CAP(T) = TABLE M(MAX,MAX)

  *
  BOUNDS        COPY
  FORM BOUNDS INITBND
  S(RAW)1,UP = TABLE B(MAX,(RAW))

  *
  ENDATA        COPY
  END

```

- ¶ Large MG programs were usually hard to follow, especially in the absence of extensive documentation relating the program to the model. Hence it was still difficult to ensure that an error-free matrix was being created. Sometimes, MG programming mistakes became evident only after the optimizer gave an implausible solution.
- ¶ Devising unique names for rows and columns was often awkward, especially when limited to 6 or 8 characters.
- ¶ MGs were not readily adapted to nonlinear programming, because most general nonlinear problems could not be expressed in terms of matrix columns.

As models became larger and more complex, these concerns grew increasingly serious. It was not uncommon to find that writing, running, and debugging an MG program cost more than solving the LP (see, for example, [13]).

Report writers

Another deficiency of early LP optimizers lay in the way they presented the optimal solution. Their principal output was a fixed-format listing of rows and columns by 8-character name, with corresponding activities, prices, slacks, limits, and so forth. Working only from these listings, modelers had a hard time assessing their solutions; and the listings couldn't be shown to a modeler's clients, who were unfamiliar with the column-name conventions, and often also with the model formulation.

The answer to this dilemma was a "report writer" (RW) system that created tabular displays according to the modeler's instructions. Like a matrix generator, a report writer was controlled by a program written in a special language, and used tables of the model's indices and data. Thus most report writers were combined with matrix generators in single MG/RW systems (in particular, all the aforementioned MGs had associated RWs). The RW language was made to resemble the MG one, though its object was to create text lines and tables rather than rows and columns. Input to

the RW included the tables used by the MG and solution activities filed by the optimizer, as well as special tables used to translate the 8-character names to more understandable text.

Being bound to matrix generators, report writers suffered from the same sorts of deficiencies. They ran in batch mode; used a special file system and environment; required fairly complex programming in a special language; and worked with matrix row and column names rather than the algebraic model. In addition, they suffered by being part of the MG system rather than of the optimizer system. Costly amounts of solution data had to be computed and filed before the RW could be run; and each new tabulation required a new RW program. A modeler making experimental runs had no way of getting tabulations simply and quickly as output of the optimizer.

Present-day systems

The LP systems used today are predominantly of the design described in past tense in the preceding sections. A self-contained simplex system solves LPs in matrix form; it is surrounded by enough of an MG/RW to translate models from the more intelligible algebraic formulation, and to translate results to intelligible tabular forms.

There have been numerous refinements and improvements along the way, of course. The algorithm handles implicit bounds and other subtleties; some MGs (for example, DATAMAT [21]) let the matrix be specified by row as well as by columns; a few systems (notably SESAME [22,23]) can be operated interactively to some degree. These changes have not affected design in any fundamental way, however. The problems mentioned above are thus still very much with us.

2 A MODERN APPROACH

There are two principal aspects to what we see as a modern linear programming system.

First, problems are described to the modern system in algebraic form, using customary mathematical notation as much as practicable.

Second, the modern system is designed to take advantage of relatively new and powerful ways of using a computer, such as interactive operation, virtual memory, and hierarchical file organization.

This section introduces the modern approach and summarizes its advantages over the traditional. Subsequent sections look at the components of a modern system somewhat more closely.

Describing the model in algebraic form

Models are first written, and usually are best understood, in algebraic form. Ideally, then, an LP system would read the modeler's algebraic formulation directly, would interpret it, and would then generate the appropriate matrix.

This ideal is beyond the abilities of current-day computers. But a modern system can come significantly close to it -- by employing a variant of algebraic form that is designed to be read by a computer system.

We call this machine-readable algebraic form a modeling language. It differs from common algebraic form mainly in employing standardized notation and terminology, and in requiring typewritten equivalents of subscripts and the Σ (sum) operator. We have devised a particular modeling language for linear programming, called XML; Figure 4 of Section 3 shows how XML would describe the model of Figure 1. (Section 3 also explains XML in more detail.)

One may think of a modeling language as the modern replacement for traditional matrix-generator languages. Both types of language are

essentially ways of describing an LP to a computer system. There are two fundamental differences, however.

First, a modeling language makes no reference to the LP matrix. It serves only to represent an algebraic form of the model. MG languages are just the opposite: they describe an LP by specifying all of its nonzero matrix coefficients.

Second, a modeling language is not a computer-programming language. It has no statements or commands, and is not run or executed. Rather, a modeling language is declarative: it serves only to describe an LP in a convenient way. A modern system reads this description, analyzes the described LP, and automatically creates the appropriate matrix. By contrast, an MG language is a programming language whose statements describe explicitly the creation of a matrix.

As a consequence of these differences, a modeling language can make the LP user's life easier in a number of ways:

- ¶ Since a modeling language looks as much as possible like common algebraic notation, it is easier to learn than a special-purpose MG language.
- ¶ Once an LP is formulated algebraically, translating it to the modeling language is essentially just a job of transcription. The work of designing and writing an MG program is eliminated.
- ¶ A modeling-language LP is as compact and as easy to understand as any algebraic description; hence it can serve as documentation of the model. By contrast, an MG program often obscures the form of the LP, and so requires additional explanation.
- ¶ Because a modeling-language LP is easily understood, it is also not hard to find mistakes or make changes. There is little of the debugging and reprogramming associated with MG languages.

¶ A modeling language identifies constraints and variables by the familiar method of subscripting. The MG's awkward concern with forming unique 8-character names is eliminated.

¶ A modeling language is easily adapted to nonlinear models: one simply permits nonlinear expressions!

In addition, once a modeling language is available, it is possible to design an interactive report-generating system that has many of these same advantages over traditional report-writers: there is no programming, language forms are easily understood, and there is no bother with 8-character names. Two such designs are outlined in Section 6.

To implement a modeling language one must build a reasonably efficient model translator: a system that reads modeling-language LPs and translates them to matrix form. The model-translator system takes the place of the traditional MG system (just as the modeling language replaces the MG language).

Can an efficient model translator be built? In the absence of a full implementation, one cannot say for sure. We have worked out what we believe to be a fairly practical design for a model translator, as described in Section 4. Further, we see ample circumstantial evidence for a workable model translator.

To begin with, the idea of having a computer read algebraic expressions is hardly new. Such expressions are essential to FORTRAN and every other general-purpose programming language, and are even used widely in MG languages (to manipulate data and calculate coefficient values). Techniques for parsing and interpreting expressions are thus highly developed, and could certainly be applied to an LP modeling language.

Going further, there are already implementations of several MG/RWs that accept algebraic constraints in limited ways. UIMP [6] specifies a model in an algebraic way, though its language differs considerably from familiar mathematical notation, and the user must still concern himself

with forming 8-character names. CML's Linear Modeling Capability [20] also has an algebraic though un-mathematical language (intended to resemble English somewhat); however, the user is relieved of naming concerns. MGG/RWG [1], on the other hand, has a fairly natural algebraic language (though the variety of expressions is severely limited), but makes the user invent a naming scheme. Most flexible is MPOS [4], which allows simple linear and quadratic expressions, written mathematically, and can translate these expressions directly to a matrix form for an associated solver; unfortunately, MPOS lacks a way to group components over indices -- every constraint must be written out individually -- and so is unsuitable for practical models of any size.

Additional evidence for the practicality of an LP modeling language comes from another sort of modeling: linear regression. Here there are also algebraic and matrix forms of the problem, and matters of data management, problem generation, and result reporting are analogous to those of LP. A modeling language for regression has been successfully incorporated in the interactive system TROLL [25,26]. It offers convenient formulation in mathematical terms, easy alteration and correction of models, quick retrieval of results, and extension to nonlinear models -- all as we envision for an LP language. Indeed, the TROLL system has been an inspiration for many ideas presented here.

Regression models are, however, fundamentally easier to describe than optimization models. The former involve few enough variables and equations that each may be named and specified individually. The latter have often thousands of variables and constraints organized in a combinatorial scheme that must be described by use of indexed subscripts. Thus TROLL's language makes no provision for subscripting, whereas an LP language requires subscripting to be practical. As a result, an LP

modeling language is definitely harder to design and implement -- which may be part of the reason that no LP equivalent of TROLL has been implemented to date.

Using the computer

A modern LP system should be an interactive one. A modeler should be able to operate the system by typing commands at a computer terminal; and the system should be able to send messages and results back to his terminal as it runs. This constitutes an interactive "session". It is in contrast to batch operation, wherein a user submits a whole set of commands -- a "job" -- and gets results only when the job is completed.

Interactive operation affords closer control of a system, and gives a better feel for what is going on. A user can "watch" an algorithm, for example, by having it report periodically on its progress. Any stored information -- a model or data, for instance -- can be immediately called up, and then changed as desired.

Interactive operation also relieves the user of many programming chores. Every batch job is actually a program (or a concatenation of several programs); the user must write out the job control statements, store them in the computer, then run and debug the job like any other program. By contrast, no interactive command need be decided upon until the previous one has finished execution. Hence a session need not be written out in advance, and mistakes can be corrected as they turn up. Interactive systems consequently encourage experimentation, which is essential to successful modeling. Ease of experimentation also makes an interactive system easier to learn.

(When a desired set of steps is known in advance, normally interactive commands can be collected as programs. Hence where programming is convenient -- in repetitive situations, mainly -- it is still available.)

What we propose should not be confused with the sort of interactive system that serves only to edit and submit batch jobs. A truly interactive user must have sufficient memory and other computer resources at his disposal to run everything -- including large LP calculations -- directly from his terminal. Today this is a feasible arrangement, thanks to computers with virtual memory and virtual-machine operating systems.

We also envision something more than just an existing LP outfitted with a command language and placed under an interactive operating system (as, for example, SESAME/DATAMAT [21,23] is an interactive MPS III/DATAFORM [18,19]). A modern LP should be designed with interactive operation in mind from the beginning. The result will differ in many ways from currently familiar LPs, as we show in the sequel.

File organization is an independent concern of great importance. Current systems produce model files, matrix files, solution files, and so forth, which the user is left to name and organize. A modern LP can take advantage of the idea of a hierarchical file system so that models are routinely linked to their data and solutions in a useful way. This idea is expanded upon in Section 4.

Outline of a modern LP

Given the above objectives, what might a modern LP system look like?

Briefly, we envision a collection of five interactive subsystems:

Model editor -- creates and updates model descriptions written in the LP modeling language.

Data editor -- stores and updates index sets and numerical data for models created by the model editor.

Model translator -- converts a model and data to matrix form suitable for an LP algorithm.

Solver -- finds (usually optimal) solutions of an LP.

Solution reporter -- builds and prints tables (and possibly graphs) of LP data and solution values.

We assume that a main interactive environment is created for the user by the computer's operating system. Each LP subsystem then makes its own subenvironment, in which its particular commands are recognized.

These subsystems may be invoked individually from the main environment, as they are required. (Not all need be used in one session.) Or, one LP subsystem may be invoked from the subenvironment of another -- for example, it will be convenient to invoke the solution reporter from the optimizer, to get tabular displays of a solution just obtained. It will also be possible for LP subsystems to invoke -- or be invoked by -- other programs outside the LP; this affords the flexibility needed to incorporate the LP in larger schemes (see, for example, [10]).

To lend some concreteness to our ideas we have worked out specific designs for each of these subsystems, to varying degrees of detail. This prospective modern system we call AMPS (Advanced Mathematical Programming System). Its constituents are XME (model editor), XDE (data editor), XMT (model translator), XLP (solver), and XSR (solution reporter); all employ XML, our version of a modeling language.

Where appropriate in succeeding sections, we use AMPS as an example to make the discussion clearer. However, we don't believe that every design for a modern system must resemble AMPS. Indeed, not all concepts of a modern system need be implemented at once; for example, a modeling language could be used with a traditional-design optimizer -- or a modern optimizer could be hooked to a traditional matrix generator.

Since our designs rely on the existence of a modeling language, we turn to modeling languages in the next section. Particulars of the subsystems are then considered in Sections 4-6.

3 A MODELING LANGUAGE

Looking back to the algebraic-form LP in Figure 1, one sees that it has five distinct parts:

- ¶ Definitions of sets of objects (over which other parts of the model are indexed).
- ¶ Definitions of parameters of the model (symbolic names for numerical data that the model requires).
- ¶ Definitions of variables in the model.
- ¶ Specification of constraints on the variables, in the form of equalities and inequalities in the variables and parameters.
- ¶ Specification of an objective, as an expression in the variables and parameters.

Our modeling language, called XML, describes a model in the same overall way. An XML model has five parts, or sections: sets, parameters, variables, constraints, and objectives. In each section, individual entities -- sets, parameters, and so forth -- are defined or specified; these entities are the section's components.

We give a section-by-section introduction to an XML model below, relating the algebraic form of Figure 1 to its XML equivalent which is shown in Figure 4. There follow a few notes on language syntax; in general, however, we have chosen to play down syntactic details, which may be found in [8].

Sets

A set is a collection of objects with which a model deals. In typical models, one finds sets of products, factories, energy sources, industries, time periods, cities, or the like. Most parameters, variables, and constraints are indexed (subscripted) over one or more sets; consequently,

Figure 4. Linear program of Figure 1a expressed in the modeling language XML. Name elements are to left, others to right as indicated. (XML does not distinguish between upper and lower case letters; the model is shown in mixed case to make it easier to read.)

SETS

prod	COMMENT:	set of products
raw	COMMENT:	set of raw materials

PARAMETERS

time	ATTRIBUTES:	positive, integer
	COMMENT:	number of production periods
max	ATTRIBUTES:	positive
	COMMENT:	maximum total unit production per period
a	INDEXING:	OVER raw, OVER prod
	ATTRIBUTES:	nonnegative
	COMMENT:	a[i,j] is units of raw material i needed to produce one unit of product j
b	INDEXING:	OVER raw
	ATTRIBUTES:	nonnegative
	COMMENT:	b[i] is maximum initial stock of raw material i
c	INDEXING:	OVER prod, FROM 1 TO time
	COMMENT:	c[j,t] is the expected profit per unit of product j in period t
d	INDEXING:	OVER raw
	COMMENT:	d[i] is storage cost per period per unit of raw material i

VARIABLES

x INDEXING: OVER prod, FROM 1 TO time
 ATTRIBUTES: nonnegative
 ALIAS: production
 COMMENT: x[j,t] is production of product j in period t

s INDEXING: OVER raw, FROM 1 TO time + 1
 ATTRIBUTES: nonnegative
 ALIAS: stock
 COMMENT: s[i,t] is stock of raw material i at
 beginning of period t

CONSTRAINTS

limit INDEXING: t FROM 1 TO time
 ATTRIBUTES: gub
 SPECIFICATION: $\text{SIGMA } j \text{ OVER prod } (x[j,t]) \leq \text{max}$
 COMMENT: Total unit production per period must not
 exceed max.

init INDEXING: i OVER raw
 ATTRIBUTES: bound
 SPECIFICATION: $s[i,1] \leq b[i]$
 COMMENT: Stock for period 1 must not exceed maximum
 allowed.

bal INDEXING: i OVER raw, t FROM 1 TO time
 SPECIFICATION: $s[i,t+1] = s[i,t] - \text{SIGMA } j \text{ OVER prod } (a[i,j]*x[j,t])$
 COMMENT: Stock for next period equals stock for
 present period less raw materials used in
 present period.

OBJECTIVES

profit ATTRIBUTES: maximize
 SPECIFICATION: $\text{SIGMA } t \text{ FROM 1 TO time } (\text{SIGMA } j \text{ OVER prod } (c[i,j]*x[j,t]) - \text{SIGMA } i \text{ OVER raw } (d[i]*s[i,t]))$
 COMMENT: Maximize total over all periods of expected
 profit less storage cost.

an XML model must specify explicitly all the sets that it uses.

The members of an XML set may be represented either as integer numbers, or as arbitrary character strings. The latter are used for any collection of named objects; a set of energy sources, for instance, might have the following members:

'COAL' 'OIL, DOMESTIC' 'OIL, IMPORTED' 'GAS' 'SOLAR'

Integer members usually represent successive periods of time. In a multi-period energy model, a set of five-year planning periods might have these members:

1980 1985 1990 1995 2000 2005 2010

Any set member may also be given an alias: a (usually) longer character string that can be used in finished reports.

In the examples above, all members are single. One may also define sets of ordered lists of any length. A set of ordered pairs, for example, can describe the arcs of a network; or, given a collection of factories and a collection of products, the collection of feasible production activities may be a set of (factory,product) pairs.

The members of any set may be given explicitly in the XML model. Usually, however, the model will only declare that certain sets exist (as in Figure 4); sets' members will be read in later as part of the model data (see Section 4). Under this arrangement a single XML model may give rise to many LPs that have the same general structure but different set data. And many common changes -- for example, adding a product or raw material, deleting arcs from a network, or altering the number of planning periods -- involve changing set members but not the model structure.

To distinguish a model's sets, they are given distinct names. The sets in Figure 4 are named PROD and RAW.

Parameters

Parameters represent the numerical data of the model: input-output coefficients, flow capacities, profits per unit of production, and so forth.

A few parameters of a model may be individual, scalar quantities: the number of planning periods (Figure 4), the discount rate, the maximum flow on any arc. Most parameters, however, are grouped as vectors, matrices, or other sorts of arrays.

A group of parameters in XML can be indexed over any set, so that there is one parameter in the group for each set member. For example, the vector B of initial stocks in Figure 4 is a group of parameters indexed over the set of raw materials. In a network model, arc capacities will be a group of parameters indexed over some set of ordered pairs of nodes.

A matrix or higher-dimension array is described as a group indexed over two or more sets. Thus in Figure 4 the matrix A of raw inputs needed for various products is a group of parameters indexed over the set of raw materials and the set of products. A conventional input-output matrix is a "square" group of parameters indexed two ways over the set of production activities. A matrix of demands for airline service in successive years might be a group indexed over three sets: departure cities, arrival cities, and years. (Three-dimensional tables are thus handled straightforwardly in XML, whereas most existing MGs restrict tables to two dimensions.)

One writes an XML indexing-expression to indicate what sets a group is indexed over. For a group indexed just over set RAW, the indexing-expression is, naturally,

OVER RAW

A group indexed over RAW and PROD is indicated by a composite expression:

OVER RAW, OVER PROD

For indexing over a set of consecutive integers, a second form of indexing-expression is more convenient; for example,

FROM 1 TO TIME

This says that indexing is over the integers from 1 to the value of parameter TIME, inclusive. By combining and generalizing these two forms, XML also makes it easy to specify more complex sorts of indexing, such as:

- ¶ Over successive integers with an increment greater than one (1980, 1985, ..., 2010, for example).
- ¶ Over all combinations of members of a set and integers in a sequence (see parameter C in Figure 4).
- ¶ Over only indices that fulfill an indexing-expression and that satisfy a specified logical condition.

XML refers to a particular parameter in a group by subscripting, as in algebraic notation. Thus, each group has a name (A, B, C, and D are the parameter groups' names in Figure 4); to specify a particular parameter, one subscripts the group name by a member or members of the indexing sets. Subscripts are listed parenthetically after the name, in the manner of FORTRAN. Following are some examples that might arise from Figure 4:

B['IRON ORE']	maximum initial stock of iron ore
A['IRON ORE','STEEL']	tons of iron ore required to produce a ton of steel
C['STEEL',3]	expected profit per ton of steel in period 3

The actual numerical values of the parameters are usually not given in the model. Parameter values are filed separately (along with set members) as model data, an arrangement that simplifies data handling and management (see Section 4) and keeps the algebraic statement of the model uncluttered. It is often valuable, however, to place in the model general restrictions on certain parameter values: that they be positive,

nonnegative, integer, or the like. Examples of restrictions like this are seen in Figure 4; they serve to catch invalid data early, before much time is wasted trying to solve an invalid LP.

Variables

The variables of a model are defined in much the same way as the parameters, either singly or (most often) in indexed groups. Each group of variables is given a name, and individuals within the group are identified by subscripting as described previously. Thus Figure 4's model has two groups of variables, X and S, whose members might include:

X['STEEL',5]	production of steel in period 5
S['IRON ORE',6]	stock of iron ore at start of period 6

A few general and common kinds of restrictions -- nonnegative, nonpositive, and such -- may be applied to a group of variables as part of their definition. (For integer programming, variables could similarly be defined as integer or real.)

Constraints

A constraint in XML is essentially just a transcription of an algebraic constraint expression. For example, suppose there are variables x and y constrained by $3x + b \leq y$ (where b is a parameter); in XML one just writes:

$$3 * X + B <= Y$$

Any linear equality or inequality is a valid constraint in XML, provided its variables and parameters have been defined in the model. (In particular, variables need not be gathered on the left-hand-side, nor constants on the right.) Nonlinear constraints could also be allowed, if computer codes were available to handle them.

Most constraints are not defined singly; they are, like parameters and variables, specified in groups indexed over sets. For example, in Figure 1, the initial stock constraints are $s_{i1} \leq b_i$, for all $i \in R$. This

is expressed in XML by defining a group of constraints; the group's indexing-expression is

$$I \text{ OVER RAW}$$

and the inequality is transcribed as

$$S[I,1] \leq B[I]$$

The name I in these expressions serves as an XML index which has exactly the same purpose as the dummy subscript i in the algebraic constraint.

Many constraints involve indexed sums, indicated by use of the familiar Σ notation. For example, the balance constraint of Figure 1 contains the term $\Sigma_{j \in P} a_{ij} x_{jt}$. This, too, is simply transcribed to XML:

$$\text{SIGMA } J \text{ OVER PROD } (A[I,J] * X[J,T])$$

The $J \text{ OVER PROD}$ in this term is just an XML indexing-expression. Any other form of indexing-expression may also be used after SIGMA -- hence most any variety of indexed sum can be expressed in XML.

Certain kinds of constraints are handled specially by the simplex method. Simple upper bounds on variables, for example, need not be carried as separate rows of the matrix if special pivoting rules are used. In an MPS deck or MG program one defines special "bound sets" to represent simple upper bound constraints and to distinguish them from constraints handled in the regular manner. XML, by contrast, specifies all bound constraints in their usual algebraic form (see, for example, the initial-stock constraint in Figure 4). It is the job of the model translator (Section 4) to interpret these constraints specially for the algorithm. An ideal model translator will pick out upper-bound, generalized-upper-bound (gub), or other special constraints automatically; alternatively, the user may identify special constraints explicitly within the XML model (as for the INIT and LIMIT constraints in Figure 4).

It is important to be able to refer to particular constraints,

especially for report writing. Thus, as with other components, each group of constraints is given a name, and constraints within a group are distinguished by subscripting the name.

Objectives

Any expression in the variables and parameters may be defined as an objective to be maximized or minimized. For a straight LP, objectives must be linear in the variables; but they may be quadratic or otherwise nonlinear when appropriate codes are available.

XML defines objectives in almost exactly the same way as constraints; the major difference is that the expression for an objective has no relational operator. Also objectives, unlike constraints, are much more likely to be single than grouped.

XML syntactic forms

It can be seen from Figure 4 that each definition of an XML component, or group of components, is composed of standard parts: name, indexing, specification, and so forth. These parts are the elements of a component. There are six different elements that a component definition can have; they are listed and briefly described in Figure 5. For each type of component certain elements are required, certain are optional, and a few may be unused.

Each type of element has an XML syntax. An indexing element must be an indexing-expression, for instance, and a specification element of a constraint must be an algebraic equality or inequality. A full and precise syntax for all XML elements is given in [8].

The headings in Figure 4 -- SETS, PARAMETERS, etc. and the element types -- are not part of XML; they have been added to make the figure readable. It would not be hard to append these words to XML, so that

Figure 4 would be a sort of "XML deck" analogous to the traditional MPS deck. We believe that a modern system will dispose with "decks" entirely, however, in favor of arrangements better suited to interactive editing; this view is expanded upon in the following section.

Figure 5. Elements of an XML component.

<u>Element type</u>	<u>Purpose</u>
NAME	Identify a component or group of components.
INDEXING	Specify how a group of components is indexed.
ATTRIBUTES	Categorize components in various ways.
SPECIFICATION	Give an expression for a component or group of components.
ALIAS	Give an alternative name to a component or group.
COMMENT	Annotate the model.

4 PREPARING AN LP FOR SOLUTION

The first job of any linear programming system is to get an LP ready to be solved. This process has three essential aspects:

First is model organization: arranging models, data, and other LP information in a consistent and convenient way. Since an LP system is largely concerned with transforming and reporting information, this aspect is a central one. Indeed, it seldom gets the attention it deserves.

Second is model creation: building descriptions of models and their data in the computer, and making corrections and changes to these descriptions. This process can be involved and time-consuming, especially when many cases are being investigated or when changes are frequent. However, this aspect also often gets slight attention.

Third is model translation: interpreting model descriptions to produce LP matrices for the simplex algorithm.

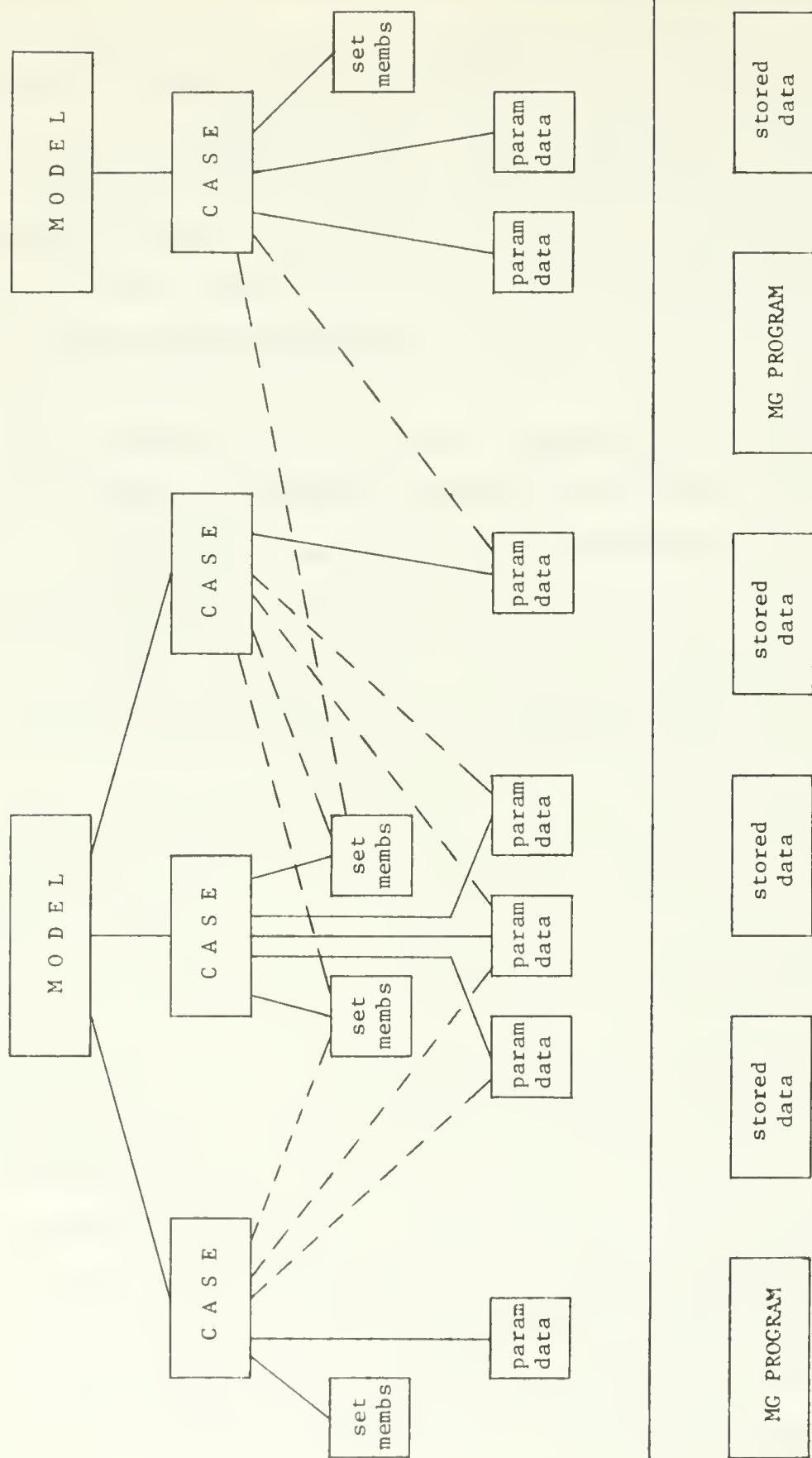
These aspects are combined in the traditional MG system. Our prototype design AMPS, by contrast, deals with each aspect independently as described below. The result is a modern system that can be more flexible and yet less complicated than MGs of today.

Model organization: a hierarchical arrangement

Traditional matrix generators are matrix-oriented: each MG program generates one particular LP matrix. Typically, an MG program begins with statements that explicitly specify all index sets and parameter tables (in Figure 3, these are the statements under headings DATA and DICTIONARY). Then come statements that build a matrix from these data.

In practice, linear programming is more nearly model-oriented. Given a knowledge of the sort of data available, one first constructs

Figure 6. Hierarchical arrangement of LP information, contrasted with a typical one-level arrangement. Dashed lines represent links.



an LP model: an abstract algebraic description that uses symbolic names for sets and parameters. A model describes an entire class of LPs, each with a different associated matrix. One then specifies any particular LP (and matrix) in that class by giving explicit data values for the symbolic sets and parameters.

Reflecting this situation, we see the XML model description as central to AMPS' organization. An XML model is thought of as being created first. Then one or more cases of explicit set and parameter data are created for the model.

Consistent with this view, AMPS organizes LP information on several levels. At the top are (XML) model-files; directly below each are one or more cases. Below each case, in turn, are data-files for its sets and for its parameters. This hierarchical arrangement is shown schematically in Figure 6, contrasted with the one-level arrangement of files of a typical LP. (Matrices, bases, and other information also fit naturally into this hierarchy, as we will show.)

A hierarchical arrangement of files is best implemented with what is known, naturally, as a hierarchical file system. The design of such a file system is a topic in itself; there have been successful implementations in a number of operating systems, including those of Multics [27] and TROLL [25,26].

One important feature of a hierarchical file system, for our purposes, is its ability to define links to a file: these make a single file appear to be in two or more places at once. A single data-file may thus be linked to several cases -- possibly under several models -- though only one actual copy of the file resides in storage. (Figure 6 illustrates this arrangement.)

Links in the hierarchy are of considerable importance for practical

work. Most models have a number of cases that differ in only a few sets and parameters; many models exist in several versions that use the same (or nearly the same) data. Thus the number of links in an AMPS hierarchy is likely to be quite large. If there were no linkage, it would be impossible to maintain the hierarchy without storing numerous duplicates of the data-files.

By contrast, an MG program incorporates both model and data in its description of one particular LP matrix. To maintain numerous versions and cases of a model, one needs a separate MG program for each, which entails considerable duplication of both the statements that define the data and the statements that generate the matrix. There are a few approaches to reducing this duplication: some MGs can store data tables in a compact form for recall by different programs; or MG programs may be built from separately-filed pieces, some pieces common to many programs and some changing from program to program. In any event, however, the job of devising and enforcing an organizational scheme falls largely upon the modeler. He may simulate a hierarchical arrangement, but the MG system cannot build a hierarchy for him.

Model creation: the model editor

Since the XML model description is central to AMPS, a user's first concern is storing his XML model in the computer. For this purpose we have designed an interactive model editor, called XME. As the term implies, a model editor is much like the sort of text editor familiar to most users of interactive systems. The difference is that, while a text editor builds a file line by line, XME builds an XML model component by component and element by element.

XME's principles of operation are exactly those of most text editors. The XME system is called to edit one model file at a time;

it may be told to work on an existing model, or to start building a new one. Upon its invocation, XME creates its own interactive subenvironment, in which its own editing commands are recognized.

The commands of XME are syntactically simple and straightforward. To add a parameter named D and specify its indexing and attributes elements, for example, one may type

```
ADD PARAMETER D: INDEXING OVER PROD: ATTRIBUTES NONNEGATIVE;
```

To replace the indexing by OVER RAW:

```
REPLACE PARAMETER D: INDEXING OVER RAW;
```

To delete the attributes element:

```
DELETE PARAMETER D: ATTRIBUTES;
```

To display parameter D at the terminal, as edited:

```
TYPE PARAMETER D;
```

To file the model:

```
FILE;
```

For convenience, most keywords can be abbreviated (PARAMETER as PAR), and repetitive phrases (PARAMETER D above) can often be omitted.

Figure 7 lists all XME commands and their functions. Complete specifications of the command language are in [7].

Why a special model editor? Originally, the description of an LP -- whether in MPS form or an MG program -- was literally a file, of cards. A user edited his files by inserting new cards and removing old ones. Eventually, this process was automated: cards became card-images on tape, and a text-editing system handled inserts and deletes. Today the card-images are most often on a disk, and the text editor is interactive and has gained many sophisticated commands. But still the user is essentially editing cards, or "lines". (And one still speaks of an MPS "deck".)

Figure 7. XME model-editing commands.

ADD	Adds new components or expressions for new elements.
CHANGE	Makes contextual changes to XML element expressions.
DELETE	Erases a specified portion of the model.
FILE	Files the current version of the model, and ends the editor session.
GET	Copies parts of another model file to the model being edited.
HELP	Types an explanation of a specified command (at the terminal).
INPUT	Adds any number and combination of new components and their elements.
LIST	Types the names of existing components (at the terminal).
PRINT	Prints specified parts of the model (on an offline printer).
QUIT	Terminates the editor session.
RENAME	Changes the name of a component.
REPLACE	Substitutes new XML element expressions for existing ones.
SAVE	Files the current version of the model.
TYPE	Types specified parts of the model at the terminal.

An AMPS user will never see a card. An XML model is not built of cards or lines, but of components and elements. Hence it is natural to design a model editor that, like XME, deals directly with the components and elements of the model. Such an editor has in fact been successfully implemented, for the stochastic modeling language of the TROLL system [25,26].

Model creation, continued: the data editor

Just as an XML model has a special structure, so do its data. Each case incorporates sets and parameters (single or grouped); sets, in turn, are composed of members, and grouped parameters are composed of numerical values arranged in indexed arrays. The job of building cases and data-files is thus analogous to that of building an XML model. For the reasons given above, this job is best done by a specialized interactive editor: a data editor, which in AMPS we call XDE.

The principles of the data editor parallel those of the model editor. XDE creates or modifies one case for one model at a time, operated by its own commands in its own interactive subenvironment. Consistent with our design approach, XDE creates cases only for existing XML models. For convenience XDE may be invoked as a subenvironment of the model editor; or, it may be called independently.

Using XDE, a modeler may specify a case by entering data values from his terminal. To do so, he would normally start by specifying sets' members. As with the model editor, command syntax is simple and straightforward; for example, to specify three members for set PROD:

```
SET PROD = {'STEEL' 'ALUMINUM' 'COPPER'};
```

Similar commands add, replace, and delete members of a set.

A single parameter is given a value by typing a simple equality:

```
TIME = 5;
```

To give values to a group of parameters, the set(s) over which the group is indexed must be specified first. Then values may be typed individually as for single parameters:

```
D['STEEL'] = 35000, ['COPPER'] = 92150;
```

or, XME can be asked to prompt for parameter values:

```
INPUT D;
```

There are also commands to change parameter values, and to add or delete values in the event that the indexing sets are changed.

Alternatively, XDE may be commanded to read data from files created outside of AMPS -- provided these files are in certain recognized formats. This option lets AMPS make use of existing data bases at a particular installation, for example.

XDE also makes limited provision for calculation with parameter values. In simple situations, however, it is clearer and less error-prone to indicate calculation in the model itself. As an example, if the model uses a parameter NET which is always the difference of parameters GROSS and EXPENSE, one may declare NET with specification element

```
GROSS - EXPENSE
```

Then XDE will only have to supply values for GROSS and EXPENSE, and NET will always be calculated correctly. (Or, one might eliminate NET from the model by substituting (GROSS - EXPENSE) wherever NET would have appeared in the constraints and objectives.) On the other hand, in complex situations -- when involved calculations are needed to derive parameters in the form the model requires, or when parameters are the output of some other algorithm (as in certain iterative schemes [10]) -- parameter values are best computed outside AMPS, by the user's own programs. These programs may call some of XDE's subroutines to store their computed values in the model hierarchy.

Finally, XDE can build a new case by linking to data files in existing cases. This option will probably be used often once a model's initial version and its first case have been created. Typically, a new case will differ from some existing case in only a few sets and parameters. Data for these few components will be typed in or otherwise supplied through XDE; but for all other sets and parameters, XDE will specify data by making links to existing cases.

Additionally, there are commands to print parts of a case for study or publication of the data, or for proofreading. These commands have the same syntax as the ones for retrieving variables' activities and other LP results; and so we describe them with other result-reporting commands in Section 6.

Preliminary specifications for XDE are forthcoming in [7].

Model translation: the model translator

One case of one XML model comprises a complete algebraic description of an LP. Once model and case are filed, there remains the task of converting them to matrix form, which we refer to as model translation.

How does model translation work? In general terms, we see it proceeding in three phases:

The first phase is syntactic and semantic analysis of the XML model; it is roughly analogous to the lexical pass of, say, a FORTRAN compiler. No case data are required by this phase.

Phase two determines what the matrix will look like: how many rows and columns it will have, and of what types. This requires the processed XML model from phase one, the case's set data, and any parameter values (such as number of periods) that affect the size and shape of the model. Output is a "model template" that associates XML variable and constraint names with their respective matrix columns and rows.

The third phase produces a representation of the LP matrix, using the model template and the case's parameter values. Upon completion of this phase, the model is ready to be solved.

There are many ways in which this outline might be realized, among them the following:

- ¶ All three phases could be run together as a single interactive subsystem.

- ¶ The first phase could be separated from the rest of the task. It might then be run before any case is defined. It could even be incorporated into the model editor, to process each element as it is typed in -- in the manner of TROLL's model editor [25,26].

- ¶ After the first and second phases, the model template could be stored permanently (as part of the aforementioned hierarchy). Then, just before the solver were to be invoked, phase three could be run to create a temporary matrix file. Since the model template can be considerably more compact than the matrix, this arrangement offers savings in storage. Also, if coefficient values change, phase three can just be run again without repeating phase two.

For AMPS, we have investigated the design of an interactive model translator called XMT.

Briefly, XMT operates in two passes: an initialization pass corresponding to phases one and two above, and a conversion pass corresponding to phase three. It is intended that both passes be called from the environment of the LP solver. The initialization pass is invoked just once. If no errors crop up, the conversion pass is then called repeatedly: each call returns one column of the LP matrix.

It is left to the solver to store the matrix columns in whatever file format it requires. Thus the design of XMT is not tied to the design of any particular solver. As a consequence, any existing LP code could readily be adapted to take its matrix from XMT.

5 SOLVING LINEAR PROGRAMS

Adaptation of the simplex method to computers is the oldest problem of LP system design, and the best understood. It is safe to say that most linear programs of current interest can -- once they are put in matrix form -- be solved satisfactorily by present-day LP codes. Hence we do not seek, in this paper, to propose improvements in LP algorithms.

Where we do see improvement is in two broader matters of design. First, we consider the external design of the LP solver: how it interacts with the user, and with other parts of the system. Second, we look at the internal design of the program that implements the LP algorithm. In both of these areas we advocate substantial departures from traditional practice.

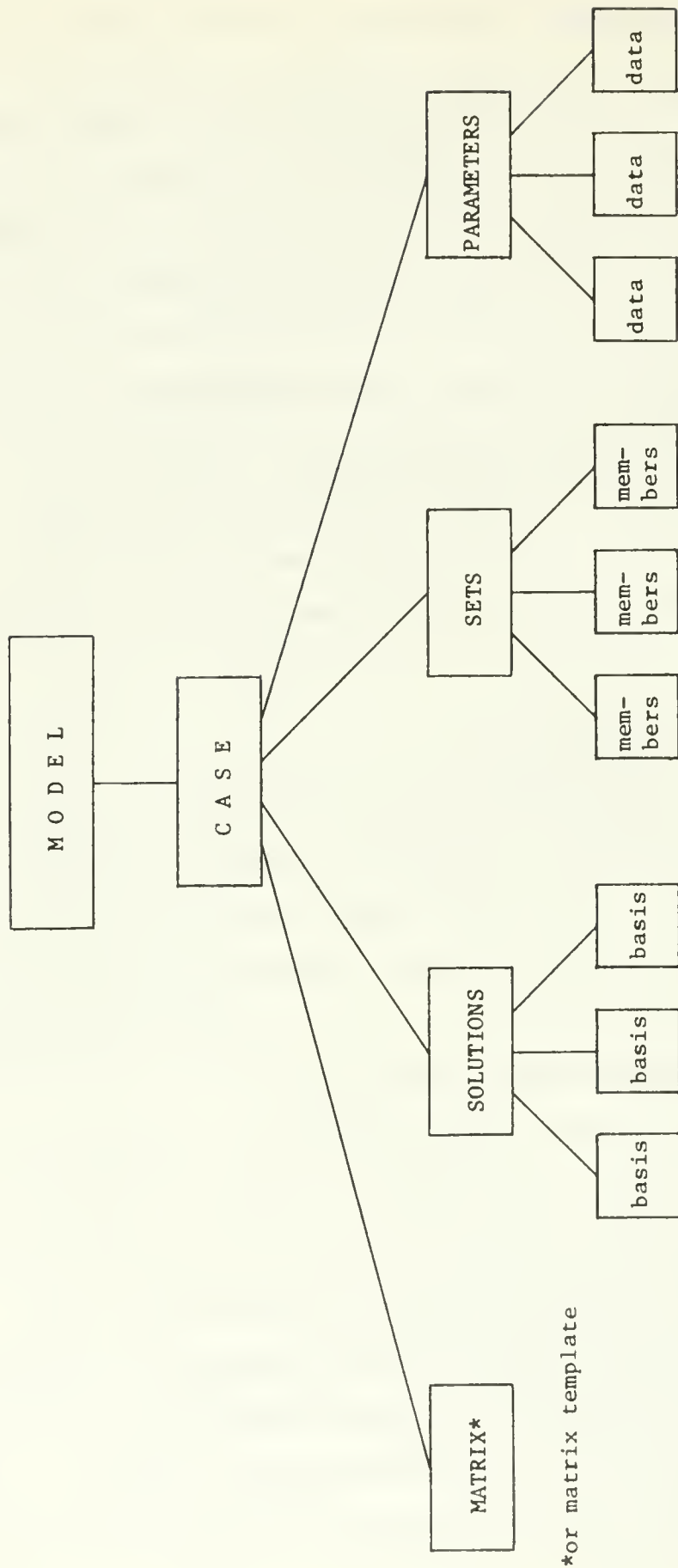
External design of an LP solver

Externally, the simplex algorithm is easy to describe. It takes a matrix and a starting basis, and produces a final, optimal basis. This description applies as well to all common variants on the algorithm -- primal phase 1, dual, parametric, crash: matrix and starting basis in, final basis out. A final basis often later serves as a starting basis, of course, for a different matrix or algorithm.

In a modern system we foresee one interactive subsystem that oversees all algorithms: the LP solver. As suggested above, its tasks are fundamentally simple: obtaining a matrix, indicating a starting basis, choosing and running an algorithm, storing the final basis. Getting an LP into matrix form is the model translator's job; displaying activities and other values associated with a basis is the job of the solution reporter (to be described in the following section).

In short, our LP solver just solves LPs. This contrasts somewhat

Figure 8. Conceptual file hierarchy for one case of one model.



*or matrix template

with traditional algorithmic systems -- MPS/360 [16,17] and its descendants [5,15,19,23], for example -- which also do a certain amount of matrix conversion and computation of solutions. (These vestigial functions date from the earliest LPs, which had as yet no MGs or RWs.)

Like other interactive subsystems, our solver creates a subenvironment in which its commands are recognized. Functionally, the commands fall into three classes -- matrix-handling, basis-handling, and algorithm control:

Matrix-handling. Most important here is a "setup" command: it specifies an LP to be solved, and arranges the LP matrix in whatever format the solver requires. Given the hierarchical arrangement of Section 4, an LP can be specified by just naming a model and one of its cases. There is then a choice of several ways to get the matrix:

- ¶ The setup command may invoke the model translator, which interprets the model and data and passes back matrix columns.
- ¶ The matrix may be read from a file. This "matrix file" is created by running the model translator in advance of the solver, and is stored under its model and case in the file hierarchy (see Figure 8).
- ¶ One or two phases of the model translator may be run in advance to produce an intermediate form of the matrix -- for example, the model template described in Section 4. The setup command then invokes the remaining phases to produce the matrix.

Generally, running more of the model translator in advance results in less work for the setup command, at the expense of more storage space. The user decides what tradeoff is least costly for his purposes.

Commands that modify the matrix will also be useful, particularly for sensitivity analyses. Possible modifications include:

changing particular coefficients; fixing or freeing certain constraints or variables; and creating new objectives or variables for parametric analysis.

Basis-handling. The solver maintains a "current" basis at all times. When an algorithm is invoked, it uses the current basis as its starting basis; it changes the current basis as it pivots and, when it stops, the current basis is its final basis.

Bases can also be stored in the hierarchy, under the model and case for which they are found (see Figure 8). Storing a basis means essentially storing a list of basic columns, possibly in a special compact form. Also stored with the basis may be additional information needed to reconstruct a solution: for example, choice of objective, values of parameters set by parametric algorithms, or changes made to the matrix after setup.

To take advantage of stored bases the solver needs a command to fetch bases from the hierarchy, and a command to save bases.

The basis-fetch command retrieves a specified basis and makes it the current basis. Any basis may be fetched, regardless of the model or case it is associated with. In particular, an optimal basis for a previous problem may be used as a starting basis for a new version. If the fetched basis is not entirely compatible with the current matrix -- due to changes in problem size or structure -- this situation is reported to the user at his terminal, and he may specify an appropriate means of recovery.

The basis-save command names the current basis and files it in the hierarchy. A basis is always saved under the current model and case (those named in the last setup command).

When the setup command creates a new matrix, it initializes the

current basis to consist of logical (slack and artificial) columns only. An all-logical starting basis is often used with a crash or primal phase 1 algorithm. In a few cases, a modeler may be able to construct a better starting basis by use of his knowledge of the problem's structure; the solver may provide basis-construction commands for this purpose.

Algorithm control. There must be a command to invoke each available algorithm. Command arguments specify particular information or options for an algorithm's operation.

General information used by many algorithms can be supplied by auxiliary commands. Examples are: choosing an objective; specifying that the basis be saved at regular intervals; indicating what information about each iteration should be displayed, and how often it should be printed.

On the whole, these solver commands are not too much different from the control statements of a traditional system. What really distinguishes a modern solver is its implementation as an interactive subsystem.

First, an interactive solver has the advantage of letting a modeler monitor the progress of his algorithm as it proceeds. He can even stop the algorithm if it runs into difficulties, or if a change in parameters or output seems desirable. "Watching" an algorithm in this way gives one a feel for how it copes with a model, and is invaluable in experimenting with a new model or algorithm. Batch systems, by contrast, give no indication of their progress -- or, more important, their lack of it -- until a run is finished; hence experimentation in batch is often costly and time-consuming, and tends to be discouraged.

Further, when an interactive solver is part of an overall interactive LP system -- such as AMPS -- it facilitates a more important sort

of experimentation: with the model itself. A user may solve a model, range or parameterize it, change it in large or small ways, and look at solution values of interest -- all during one session. Results are available immediately at the terminal. Hence a modeler's next step may be based directly on what he has just done, and errors are corrected immediately. Even quite involved modeling strategies are reduced to typing some commands, looking at the output, and deciding what to type next. In short, interactive operation encourages creative manipulation of the model to discern its properties and inadequacies.

Batch operation, on the other hand, tends to discourage experimentation in favor of a fixed plan of action. New ideas require delays while new runs are made. A complex modeling strategy requires a complex batch control program, or numerous simple runs. And whole runs are often invalidated by simple programming errors. (Of course, experimentation eventually gives way to production runs, for which batch is adequate. But production runs are just as easily done on an interactive system, by use of small command programs.)

It is also important that a modern solver be only a subsystem that can be invoked from other environments. This makes it relatively easy to incorporate the simplex algorithm in larger iterative schemes -- a common approach in integer programming, and a successful technique in finding economic equilibria. Some traditional LPs are designed to operate as dominant systems only, and can be subordinated to other programs only with difficulty.

Internal design of an LP algorithm

Both time and memory space were at a premium when the simplex method was first implemented on computers. Thus the emphasis in early LP codes was on clever programming in assembly (machine-level) languages to achieve

the greatest efficiency. Since the matrix, basis inverse, and intermediate computational results could not all fit in memory, much of the code was concerned with exchange of data between memory and external storage (tape or disk). These concerns persisted in later systems, including ones still used extensively (such as MPS/360 [16,17], MPSX/370 [14,15], MPS III [19], and APEX-II [5]).

Today's computers are faster by orders of magnitude. With the use of virtual operating systems, much or all information needed by the algorithm can be kept in memory. Thus clever design is no longer so important, and the emphasis can be shifted to convenient and reliable design.

Specifically, a modern algorithmic code should be a modular one. A module is just a subroutine whose input, function, and output are precisely defined, and whose internal design is independent of other modules' designs; a modular system is one built of such modules. A modular design for the simplex algorithm would probably have four main computational modules:

PICK INCOMING VECTOR -- Choose the next column to enter the basis, or report that none can be found (optimal solution).

PICK OUTGOING VECTOR-- Given an entering column, choose a column to leave the basis, or report that none can be found (unbounded solution).

PIVOT -- Given an entering and a leaving column, update the current basis and the representation of the basis inverse.

INVERT -- Given a basis, compute a compact representation of the corresponding basis inverse.

A simple controller module would actually run the algorithm by repeatedly calling these modules in the proper sequence. The controller would also cause requested output to be printed, and would terminate the

algorithm at the appropriate point (on finding an optimum, reaching an iteration limit, or whatever).

Each module in a modular system is itself built of sub-modules, which are in turn built of sub-sub-modules, and so forth down to the simplest level of organization. Some sub-modules for the simplex algorithm will do computation; others will read and write the matrix and other data structures. Following the principles of modular design, the computation sub-modules will be independent of the data-reference sub-modules -- in contrast to some older codes in which these functions are mixed together.

The advantages of a modular simplex code are many. For one thing, it is easier to conceive a good modular design; though the algorithm may be complex overall, it is easily dealt with once broken into small, independent parts. Other benefits go beyond the stage of design: they are realized in coding the algorithm, in using it, in modifying it, and in coding other algorithms:

Coding. It is easier to program a modular system correctly (than to program one that is not modular), since it can be coded one simple sub-module at a time. Moreover, it is easier to debug a modular system: modules can be tested out individually before they are connected, and bugs that show up in the full system can usually be traced to one or a few independent modules. Ease of debugging is especially important: bugs still turn up in LP codes that have been used for many years.

Along with correct coding should come correct documentation, which is also facilitated by modular design. The idea of a module as a fully specified unit promotes good documentation in itself. Further, it is not so hard to document simple sub-modules one at a time, whereas documenting a whole un-modular system can be a hopeless chore.

Using. An algorithm that is better built is naturally better to use. For example, any bugs that turn up are more quickly fixed.

Most important here, however, is documentation: a system that has good internal documentation also tends to be well-documented for users. In particular, it should be clear just how the simplex method has been implemented, and exactly how it is affected by settings and options available to the user. Such information is not needed all of the time, of course, but when difficulties arise it is invaluable. Unfortunately, essential information is often lacking or obscured in the documentation of traditional LPs. (One system, for example, offers optional "scaling" of the matrix, but its manual [19] nowhere indicates what sort of scaling is done!)

Modifying. Changing a modular system amounts to modifying or replacing an appropriate sub-module. Since sub-modules are independent by design, the change should have little effect on the rest of the system. Thus it should be fairly easy to try out new ideas -- a new pricing scheme, for instance, means just a new submodule in the PICK INCOMING VECTOR module.

Traditional LPs, by contrast, are highly interconnected; even a minor change risks introducing a host of new bugs. Given also their poor documentation, production LPs are frozen for all practical purposes. As a consequence, algorithmic experimentation with problems of practical size has always been difficult.

Coding other algorithms. Well-designed modules need not be confined to one algorithm. The simplex modules PIVOT and INVERT, for example, can be used as well by related pivoting algorithms (parametric, dual, or crash). Submodules of PIVOT might even serve for entirely different algorithms that use pivoting of one sort or another (certain

fixed-point algorithms, for example). Thus a library of debugged and documented modular algorithmic codes can serve as a good start for coding new algorithms.

The value of this sort of arrangement has been recognized in MPSX/370 [14,15]: fifteen of its high-level modules can be called as subroutines from programs written in PL/I. (MPSX/370 is not a fully modular system, however; it is essentially a modular adaptation of the original design of MPS/360 [17]. Its high-level modules are highly interdependent, employing a large pool of common data. Modules at lower levels are inaccessible to users.)

6 REPORTING RESULTS

The LP solver produces an optimal basis, while what the user wants are optimal activities and other solution values. Thus any LP system needs ways of storing and retrieving solutions, and of extracting solution data to make readable tables. Here we contrast some traditional and modern approaches to these tasks.

Storing solutions

A model, its data, and a basis together define a particular solution to a particular LP. For each solution there are various results: activities of variables, sums and slacks of rows, reduced costs (prices) of variables and constraints, sensitivity ranges on objective coefficients and right-hand sides, and so forth.

Results are easily generated when a model has just been solved and the optimal basis is still current. Very often, though, one wants a way of storing a solution to look at its results later. A solution can be stored effectively in any of several forms, as follows:

Model + data + basis. One can store the model and its data in their original forms, along with a list of basic columns. These suffice to recompute the results when they are wanted.

For a traditional system, this amounts to saving the MG program and basis; later the program may be re-run, the resulting matrix set up, and the basis restored. For a modern system such as AMPS, solutions are stored this way automatically as branches (model → case → basis) in the file hierarchy (see Figure 8, Section 5). Later the matrix can be recreated from model and case data by use of the model translator.

In any sort of system, this is the most compact way of storing a solution, and so is valuable when results of many runs are to be kept

available for later scrutiny. Some cost is incurred, though, in regenerating the matrix each time results are needed.

Documentation is also a concern when many solutions are stored. If a solution is stored as an XML model plus case plus basis, it is fairly easy to determine later just what version of the model it represents. On the other hand, it would also be easy to invalidate the solution by inadvertently changing the model or case data. One might therefore want a way to "freeze" a solution so that further such changes were rejected.

Matrix + basis. An alternative is to store the entire matrix -- as output by the MG or model translator -- along with the basis. From this form results are more readily calculated, but often much more storage is required. Still, storage may not be a problem if few solutions are stored, or if they are not stored for long.

This form also has its own documentation problems: it is possible to end up with a store of matrices but no record of just which version of the model gave rise to each one. On the positive side, the chances of inadvertently changing a stored matrix are slim.

Table of results. A different approach is to store all and only the result values, in some sort of comprehensive table in which they can be looked up later. This gets the calculation done once and for all, and leaves no chance that the solution will be invalidated. The price is high, however: the result table needs a huge file; much time is inevitably wasted in calculating values that are never of interest; and documentation is a sore point as for matrix + basis.

Traditional RW systems generally require a table of results as input. This practice can be traced to the fact that RWs have usually been designed as appendages to existing LP codes; a "solution file" is

thus appealing as a simple and well-documented interface. To reduce unneeded computation, RWs make limited provision for filing a subset of results; still, generally more is filed than is needed, especially by inexperienced users.

Given a well-documented, modular modern system, there is no compelling reason to have a result-table as the interface between the routines that calculate results and routines that use them. Instead, the latter routines can just call the former each time a result calculation is required. Result tables are thus unnecessary to a modern system.

Tabulating results

A more complex problem is that of displaying a solution. Simple listings of results are seldom revealing; one needs to organize and abstract result values as manageable tables. Further, it is seldom enough to look at just raw results from one solution at a time. Useful tables often include other quantities: model data, including costs and upper limits; values computed from results and model data, such as percentages of capacity used and totals of many sorts; and results from other solutions, for comparison.

To further complicate matters, there are two conflicting demands for tabulated results:

Quick tabulation -- When a model is being developed and tested, the modeler needs an easy way of getting results out fast and in a variety of forms. Considerations of format and annotation are secondary, since at this stage the table is read mainly by people familiar with the model.

Reporting -- Once a model is in routine use, its creators need carefully laid-out tables that can be printed in reports and shown to clients who don't know the model well. At this stage it is not so important to be able to generate new tables fast or easily, since the same table layouts will usually be used for many solutions with little change.

A computer system can be built to tackle either one of these requirements, but there is little hope of one system satisfying both. Any system complex enough to produce finished reports will inevitably be too complex to print simple tables fast and easily.

To deal with these requirements, traditional LP systems have given users three essentially different ways of printing results:

- ¶ LP algorithmic codes print standard tables of results, one line each for selected constraints and variables. These tables are sometimes used as quick tabulations, but are usually inadequate because their format is so rigidly fixed.
- ¶ Report-writer systems are run by programs written in special languages designed for describing tables line by line. They can produce a variety of tables and, for lack of a better alternative, are often employed in both quick tabulation and reporting. In fact, the great weakness of report writers is that they compromise between these two requirements, and so serve neither well. Writing and debugging their programs is seldom fast or simple, especially in batch operation; yet their languages are not powerful enough to produce some reports.
- ¶ Users' programs in general-purpose languages (like FORTRAN and PL/I) can read filed results and organize and print reports. These programs take time to develop, but they can produce virtually any conceivable report.

In summary, traditional systems are weakest in quick tabulation; reporting fares better, since it can always be handled by user-written programs. The only systems designed specifically to organize results -- the report writers -- are not ideal for either quick tabulation or reporting.

The modern approach to quick tabulation requires, naturally, an interactive subsystem. Results are retrieved, organized, and printed by typing commands at a terminal; there is no delay while programs are

written, debugged, or revised, and so tabulation can be truly quick. Further, interactive operation is ideal for development and testing, when one wants to look at each new table as it is typed. Also, the interactive command language can be kept fairly simple and easy to use, since only rudimentary formatting and annotation is required.

Interactive operation is not the whole solution, however. To be effective, the quick-tabulation subsystem must build and print each table as a whole -- not line by line, as a traditional report writer does. For example, there should be a single, concise command to say, "Print a table of production activities at each factory in each period, with factories running across the top and periods along the side." The system should then automatically figure out column spacing and generate all of the lines (rows) of the table. If there must be one command for each line of the table, the user is inevitably forced into programming and formatting, even if the system is nominally interactive (a good example is DATAMAT [21]).

For reporting, on the other hand, one still needs a programming language that can form and put out tables by line; there are just too many options to handle in a simple command language. We see little advantage, however, to designing a special programming language (such as an RW language) for this purpose. Any general-purpose language that can handle character strings is sufficient; moreover, it is already implemented, and may already be familiar to the modeler. In addition, if the LP system is a good modular one then there will already be well-documented subroutines that fetch data and compute results; the modeler's own program can simply call these to get the values it needs.

Tabulating in AMPS: XSR

Our AMPS system includes an interactive, quick-tabulation solution reporter called XSR. It is possible to invoke XSR from within the environment of the solver, to look at results for a basis that has just been found. In addition, one can invoke XSR independently of the solver to look at one or more filed solutions.

The XML modeling language is employed as much as possible in the command language for XSR. In this way commands can be kept short and simple: XML is a concise way of referring to the model, and it uses the same notation for data (sets, parameters) as for results (variables, constraints, objectives). (In fact, XSR can make displays of LP data even before the LP is solved.) Further, the XSR command language should be easily learned since users must already know XML.

We have investigated two quite different designs for XSR. Both embody a tradeoff between speed and simplicity, on the one hand, and power to generate a variety of tables, on the other. They differ in how the tradeoff is chosen.

The first design emphasizes speed and simplicity: an entire table can be generated and printed by typing a single command. Straight-forward default conventions enable short commands to create quite extensive tables. For example, the following command lists activity (scaled by 100), cost, and cost sensitivity for all variables in group X:

```
VARIABLE X: ACTIVITY/100, MIN_COST, COST, MAX_COST;
```

If X is indexed over two sets, an even simpler command tabulates X activities in a two-way table, the first index running vertically and the second horizontally:

```
VARIABLE X(I, BY J): ACTIVITY;
```

More sophisticated command forms select only certain components of a

group, and specify a wide variety of values for the table body.

This design handles simple tables best, while more involved arrangements are beyond its power. It also enforces a uniform layout and format, though a few auxiliary commands are provided to alter formatting defaults (such as number of decimal places printed).

The second design is more complex but also more powerful. A set of commands is used to build a table, piece by piece. Separate commands specify indexing, body, headings, and format; the table may have two or more dimensions, and may be divided into parts that have different characteristics. Once a table is built, separate commands print it in full or in part, in any specified arrangement.

Because many commands can combine to create one table, this design lets quite complex tables be built even though no particular command is especially complicated. Further, a table need not be printed just once: a series of useful "slices" might be generated from it. More work is required to print a simple table than in the first design, however.

Further comparison of these designs, and details of each, may be found in [9].

Graphical presentation of results

Much of what has been said above applies equally well to displaying results graphically -- a thoroughly neglected topic, insofar as it applies to LPs. Again, there is a question of quick tabulation versus reporting. Present-day users, however, have only the option of writing their own programs to produce graphs; these programs are necessarily fairly complex, and hence "quick graphics" is an impossibility.

XSR remedies this omission: both designs described in the previous section can be adapted to graphing by addition of appropriate commands. With the first design, a single command serves to create an

entire graph in some standard format. The second design provides graph-building commands that create an n-dimensional space of points, select projection planes through the space, and print or display the projected points as graphs.

Equivalents of both of these designs have been implemented to graph time series in TROLL [26]. In particular, the TROLL subsystem CLOUDS [24] incorporates an extensive set of graph-building commands (which were also the inspiration for the table-building commands described in the preceding section).

REFERENCES

- [1] Alsbury, P., et al., "The MGG/RWG System". Scientific Control Systems Ltd., London, 1975.
- [2] Beale, E.M.L., "Matrix Generators and Output Analyzers", in Proceedings of the Princeton Symposium on Mathematical Programming, Harold W. Kuhn, ed. Princeton University Press, Princeton, N.J., 1970.
- [3] Bonner & Moore Software Systems, "A Description of Bonner & Moore's GAMMA System". No. CSH-007, 1976.
- [4] Cohen, Claude and Jack Stein, Multi-Purpose Optimization System: User's Guide, Version 2. Manual no. 320, Vogelback Computing Center, Northwestern University, 1975.
- [5] Control Data Corporation, APEX-II Reference Manual. Publication no. 59158100, Revision C, 1974.
- [6] Ellison, E.F.D. and G. Mitra, UIMP: User Interface for Mathematical Programming. UNICOM Consultants, UK Ltd. (no date).
- [7] Fourer, Robert, XME Model Editor and XDE Data Editor for Linear Programs. Available from the author.
- [8] ———, XML Modeling Language for Linear Programming: Syntactic and Semantic Specifications. Available from the author.
- [9] ———, XSR Solution Reporter for Linear Programs. Available from the author.
- [10] Harrison, Michael J., "Software Tools for Combining Linear Programming with Econometric Models". Paper presented at the Bicentennial Conference on Mathematical Programming, Gaithersburg, Maryland, November 1976.
- [11] Haverly Systems Inc., LP 360/370 Linear Programming System: User and Operating Manual, Sixth Edition. November 1977.
- [12] ———, Omni Linear Programming System: User and Operating Manual, First Edition. October 1976.
- [13] Hogan, William W., Jr., "Matrix Generation, Report Writing, and the Scale of Future Applications". Working paper prepared for the fall joint meeting of ORSA-TIMS, Miami, November 1976.
- [14] IBM Corporation, IBM Mathematical Programming System Extended (MPSX/370), Control Languages. No. SH19-1094, 1976.
- [15] ———, IBM Mathematical Programming System Extended (MPSX/370), Program Reference Manual. No. SH19-1095-1, 1976.
- [16] ———, MPS/360 Version 2, Control Language User's Manual. No. GH20-0290, 1971.

- [17] IBM Corporation, MPS/360 Version 2, Linear and Separable Programming User's Manual. No. GH20-0476, 1971.
- [18] Management Science Systems, MPS III DATAFORM: User Manual. 1975.
- [19] ———, MPS III Mathematical Programming System: User Manual. 1975.
- [20] Mills, Ronald E., Robert B. Fetter and Richard F. Averill, Conversational Modeling Language Reference Manual. No. W7-54, Center for the Study of Health Services, Yale University, 1977.
- [21] National Bureau of Economic Research, Computer Research Center, DATAMAT Reference Manual, Third Edition. Order no. D0078, 1975.
- [22] ———, SESAME Primer, Second Edition. Order no. D0076, 1976.
- [23] ———, SESAME Reference Manual. Order no. D0077, 1975.
- [24] ———, TROLL Experimental Programs: Interactive Data Analysis, Econometrics, and Numerical Methods. Order no. D0070, 1974.
- [25] ———, TROLL Primer, Second Edition. Order no. D0046, 1975.
- [26] ———, TROLL Reference Manual (Standard System). Order no. D0062, 1976.
- [27] Organick, Elliot I., The Multics System: An Examination of Its Structure. MIT Press, Cambridge, Mass., 1972.

